

DSt

Datorsystemteknik
Department of Computer Engineering
University of Lund



Aspects on Implementation of Ada[®] on Multiprocessor Computers with Shared Memory

Anders Ardö

Lund 1986

to Ylva



To keep an
ever-open door
is wisdom's true advancer;
so they are fools
who don't ask more
than ten wise men can answer.

Piet Hein

Aspects on Implementation of Ada[®] on Multiprocessor Computers with Shared Memory

Anders Ardö



Digitized by the Internet Archive
in 2026

https://archive.org/details/IA41546727_0068

1 Introduction

During the past few years several high level languages with mechanisms for parallel programming have been developed, e.g. Concurrent Pascal, Modula-2, Chill and Ada. Even though they were designed to express parallelism, they are mostly intended to be implemented and used on ordinary computers with only one central processing unit. Parallel programming is then used as a convenient way of expressing logical relationships between different parts of a complex program.

But in the case of multiprocessor computers such languages become essential in order to handle the actual parallelism in the hardware. As there are still very few genuine multiprocessor computers in practical use, rather little is known about the implementation of parallel languages on such machines.

The use of high level parallel programming languages is becoming more and more important. Many computational problems have an inherent parallelism that obviously will be utilized only when parallelism has become part of the programmers' daily life. Using parallel processes is a powerful structuring tool that increases program understandability. Real time programming is an obvious example for which these arguments are relevant, but as experience grows the same will show to be true for a variety of other application areas.

There are several reasons to use Ada for a study like this. Ada is one of the very few standardized parallel languages. One can also suppose that it will be widely used also in multiprocessor environments. The type of applications for which Ada is likely to be used are also the kind of applications anticipated for multiprocessor computers.

This thesis considers various aspects of implementation of high level parallel languages, specially Ada, on true multiprocessor architectures with shared memory and is based on the following papers:

- I. A. Ardö, L. Philipson: *"Implementation of a Pascal Based Parallel Language for a Multiprocessor Computer"*, Software – Practice and Experience, vol 14(7), pp 643-657, July 1984.
- II. A. Jones, A. Ardö: *"Comparative Efficiency of Different Implementations of the Ada Rendezvous"*, Proc. of the AdaTEC conf. on Ada, pp 212-223, Oct 1982.
- III. A. Ardö: *"Experimental Implementation of an Ada Tasking Run-time System on the Multiprocessor Cm*"*, Proc. of the 1st Annual Washington Ada Symp., pp 145-153, March 1984.
- IV. A. Ardö, L. Philipson: *"A Simple Ada Compiler Invalidation Test"*, ACM Ada LETTERS, Vol III, no 5, pp 69-74, April 1984.
- V. A. Ardö: *"Experience Acquiring and Retargeting a Portable Ada Compiler"*, Technical Report, Dept. of Computer Engineering, University of Lund, 1986.
- VI. A. Ardö: *"Efficiency Aspects on Ada Run Time Support for Multiprocessors with Shared Memory"*, Technical Report, Dept. of Computer Engineering, University of Lund, 1986.

2 Early experiments

Paper I describes the implementation of a Pascal based parallel language on a small experimental multiprocessor computer built at our department (Philipson et al 1983). The interesting part is not so much the language, but the implementation technique used, similar to the one later used in the project reported in paper V.

This project was defined in 1979. At that time few, if any, implementations of parallel languages on multiprocessors were reported in the literature, and the project that later led to the definition of Ada was still in an early stage. Our implementation and the early experiments with parallel programming of a multiprocessor computer provided the necessary background for the subsequent work on Ada.

We chose to base the parallel language on the Pascal P4 compiler originally written at ETH, Zurich (Nori et al 1981). The implementation was based on cross compilation, code generation and linking on a host computer, from which the executable code can be downloaded into the target microprocessor computer.

The small multiprocessor used for this experiment is composed of a number of processing elements connected to a common, global bus. All processing elements are identical, except for a unique identification. Each processing element contains a CPU, a part of the global memory, some private memory, and I/O. The global bus of such a system is a potential bottleneck, so we increased the capacity of the bus by using not a single bus, but a set of parallel busses. In our case the global bus consists of three parallel channels in a way that is completely transparent to the processing elements.

The P4 compiler is a portable front-end compiler for a large subset of 'standard' Pascal. The front-end compiler does not generate executable code directly, instead it generates an intermediate representation of the program and does all the syntactic and semantic analysis. The P4 front-end compiler generates P-code. This P-code is input to a back-end which generates executable code for the target. This technique makes it possible to use the same front-end together with several different back-ends to produce code for several different target machines. It is also possible to use different front-ends producing the same intermediate code with a back-end thus making several different languages available for a target machine.

We made all the extensions at the code generation level and did not change the syntax of the Pascal language. In fact the technique used here made it possible to use the front-end part of the compiler without any changes. This was crucial in our case since at the beginning of the project we did not have the possibility to recompile the front-end itself. Furthermore, all extensions were concerned with the interaction of parts of a program executing in parallel. By altering the meaning of some of the P-codes we were able to introduce parallelism in the language.

The two principal extensions for communication and synchronization between processes (modules) were *external procedures* and *semaphores*. Both of these use an underlying hardware mechanism (exclusive-access) to provide protection for control data structures. The semantics of an external procedure call supports two different parallel constructs, fork and the Ada rendezvous.

A program for the multiprocessor computer consists of one or more modules. Each module is coded as a complete Pascal program, possibly with some procedures declared as external. Each processing element of the multiprocessor can contain only one module. Which processing element it is actually loaded into is determined

dynamically at the time the entire program is loaded.

This experiment showed that even with a very limited multiprocessor system, interesting experiments with implementing parallel programming languages can be performed. It gave valuable experience of multiprocessor specific issues, as basis for the further studies concentrated on Ada.

3 Experiments with efficiency of the rendezvous

One reason to build and use multiprocessor computers is of course to increase the capacity of one machine. Therefore it is crucial to utilize the resources of the machine in an efficient way. Paper II attacks this problem for process synchronization in Ada.

Reasonably efficient implementations of Ada task synchronization, in particular the rendezvous, will be crucial to the successful application of Ada. There are a variety of different implementations of rendezvous semantics, each with a different cost. In addition, some implementations can only be used under constrained circumstances. In paper II a particular application of the rendezvous is chosen that we expect to occur frequently. Several alternative implementations for this use of the rendezvous were programmed and the performance of these implementations on the shared memory multiprocessor Cm* (Jones and Schwarz 1980) were measured to ascertain actual costs of the various implementations on a multiprocessor computer.

Cm* is a multiprocessor composed of 50 computer modules each consisting of a Digital Equipment Corporation LSI-11, a standard LSI-11 bus, memory and peripheral devices. All primary memory in the system is accessible by each processor through a distributed switch. Cm* was designed and built at Carnegie-Mellon University, Pittsburgh where this work was done.

To make actual measurements we had to select a specific use of the rendezvous. We had no Ada compiler for Cm*, so the application had to be simple. We wanted to be able to make reasonable judgements about possible implementations, because code would be generated by hand, not by a compiler. We also wished to measure the cost of providing rendezvous semantics for routine, rather than esoteric, uses of the rendezvous.

The application we chose is an example of what is usually referred to as the server model. A homogeneous set of tasks is responsible for processing a variable-sized set of work requests. By homogeneous, we mean that any task, which we will refer to as a worker, is capable of processing any work unit. Work units may be statically known or may be dynamically generated by a master. We expect the server model to be used in the implementation of many parallel algorithms. The server model has been heavily used in both operating systems and applications on Cm*.

We hand coded an implementation of the server model using three different strategies for synchronization and task intercommunication (rendezvous implementation):

- **message transmission** – The MASTER and the WORKER's send short data messages via a FIFO message buffering structure to communicate both a WORKER's request for a new work unit and the MASTER's assignment of a specific work unit to satisfy the request.
- **busy wait on shared variables** – the MASTER and WORKER's share

variables, busy waiting on them when necessary. No locking of shared variables needed.

- **in-line embedding** – there is no MASTER task; assignment is performed in-line by WORKER tasks. Locking is used to synchronize use of the assignment data structures.

The fourth strategy does not implement the above requirements, but is an efficient implementation of the server model with the assumption that assignment can be done statically. It is included to act as control against which to compare our experimental results.

- **static assignment** – each WORKER knows from the beginning the sequence of work units it is to process. No synchronization or communication among tasks is required.

All implementations were hand-coded and are representative of the highly optimized code that a production quality compiler might generate for the specific example. We believe that there is an opportunity for an optimizing compiler to recognize particularly simple uses of the rendezvous mechanism, and then to select among several different implementations of the rendezvous. This is certainly the case when an application is to run on a distributed system. It is unlikely that the most efficient way to implement a rendezvous within one processor is most efficient for a distributed system.

We showed that synchronization and communication costs associated with assigning work units to a set of tasks capable of processing the work units vary considerably across different implementations. The most expensive strategy (the message transmission) is roughly 18 times more expensive than the most efficient strategy. Static allocation incurs the lowest cost, but its applicability is constrained. Among the implementations that assign work units to WORKER tasks dynamically, a global shared data structure – not involving locks – is the most efficient. The locking scheme that we measured incurred increasing cost due to contention as the number of processors involved increased.

These observations are based on the assumption that the cost to actually process a work unit is negligible. As the cost to process the assigned work unit increases, it will eventually dominate the synchronization and communication costs of assigning a work unit to an individual task. The most interesting observation for this case is that when less than 10 processors were involved, strategies that dedicate a task as a global master who assigns work to other tasks perform less well than strategies that allow all tasks to process work units. As the number of processors is increased beyond 12, the cost difference were not noticeable.

4 Tasking implementation

It is not only the efficiency of the rendezvous that determines how efficient Ada will execute on a particular machine. The semantics of Ada can in some cases cause considerable overhead in the run-time system of a particular machine if the architecture of the machine does not fit Ada well. In paper III, an Ada tasking run-time system is reported. In that paper the semantics of Ada tasking is analysed and

some mismatches between a message based architecture and the rendezvous model are located.

Efficient and reliable implementation of full Ada tasking on multiprocessor computers raises several new issues compared to uniprocessor implementations. There is no reason to believe that a uniprocessor implementation will function effectively on distributed hardware. In particular it cannot be expected to exploit the potential for parallelism and robustness in distributed architectures, or to deal effectively with the distributed resources. The run-time system reported in paper III implements almost full Ada tasking, thus demonstrating the feasibility of using Ada on a true multiprocessor. The implementation was done on the multiprocessor computer Cm*.

Even if Ada was designed to be used for embedded computer systems, no special care has been taken to make multiprocessor implementations efficient. On the contrary the rendezvous, being the only synchronization mechanism, is a large and complicated construct which requires considerable overhead to implement with its full semantics. In a direct and naive implementation, like the run-time system described in paper III, it takes about 100 LSI-11 instructions to execute a complete rendezvous, in the **most favourable case**. This figure does not include any statements executed within the rendezvous but only the synchronization and bookkeeping needed to perform a rendezvous.

Not all multiprocessor architectures are suitable for Ada. An architecture that uses a pure message based scheme as basis for synchronization and communication is badly suited for an Ada implementation. One problem is of course the need in Ada for a common address space shared between the tasks. The shared address space is used for shared variables and parameter passing between tasks in entry calls. But the main problem for a pure message based implementation has to do with the fact that entry calls can time out and thus the caller has to cancel the call, i.e. remove the entry call from the queue. The synchronization needed to do this removal indivisibly with respect to the acceptance of the call by the called task is difficult to achieve within the message model. Of course there are ways to circumvent the problem but they seem to be too costly in space or time or both.

Paper III concludes that an implementation using shared memory as basis for synchronization and communication is more efficient than a message based implementation. The use of shared memory provides no problems in these respects. Rather it was easy to map Ada onto traditional shared memory primitives. An implementation and evaluation of the memory management involved in supporting separate stacks (the cactus stack) for each task is reported in paper VI. The run-time system uses shared memory for communication, i.e. all shared variables, task control blocks and status flags are located in the shared memory and thus available to all tasks. The lowest level synchronization primitive used is an operating system provided lock/unlock mechanism. The use of these traditional shared memory primitives at the lowest level of the run-time system made it fit easily onto the model of the world that the underlying operating system uses.

5 Full Ada implementation

It turns out that implementing a full Ada compiler and run time system is hard to do correct and efficient. It is only recently that useful, validated Ada compilers have

become available, five to six years after the first reference manual was published and one to two years after the language definition became an ANSI standard. Even the compilers today can be improved to reach the status of good production quality compilers.

The complexity of implementing Ada correctly (especially tasking) can also be concluded from the difficulty most Ada compilers have had with the small Ada compiler invalidation program described in paper IV. Even though it has been available for more than two years, tasking bugs in validated compilers are still found with the help of this program and variations of it.

The program is a small (two pages of Ada source code) fairly simple program exercising the parts of tasking that are hard to implement. By writing one program that uses all these tricky parts, the interaction between them are tested as well. It might very well be that all different language constructs are implemented and tested one at a time, but when all of the parts are put together some unexpected interaction occurs.

For a small research group it is not possible to do a complete Ada compiler for a piece of non-standard hardware to use as a tool in a research project. There is too much effort involved to justify this. Paper V describes a technique for implementing Ada on a piece of non-standard hardware with reasonable effort.

A portable Ada front end compiler was used for retargeting. Buying the front end compiler showed to be a complex and time consuming process, partly because we did it very early and partly because of our specialized requirements. In paper V the process of acquiring and evaluating the front end is described briefly. A more detailed description of this process has been published elsewhere (Ardö and Philipson 1984). Based on this experience, comments on validation, quality and efficiency of Ada compilers are given in paper V. Among other things it is argued that an evaluation of Ada compilers must contain much more than the validation.

The target machine for our retargeting is an experimental multiprocessor of the MIMD type with shared memory, based on NS32000 (Stenström and Philipson 1985).

In paper V a detailed overview of the resulting Ada system is given. The system was built from scratch on the bare hardware. As with the front end we tried to buy as many components in the system as possible. It has three main components: a code generator, a run time system and an Ada kernel.

The code generator is table driven and generates symbolic NS32000 assembler. This code is then assembled and linked by commercially available components. The run time system and the kernel is implemented in assembly language and Concurrent Euclid and handles tasking, exceptions and scheduling. The result is a complete Ada implementation that can execute one Ada program on one processing element.

The approach used here has shown to be successful. No major problems were discovered during the implementation effort. Paper V concludes with some advices for other similar projects based on the experiences gained.

6 Efficient run time support

Paper VI analyses efficiency aspects of the run time support for an Ada implementation. It starts with a brief description of the Ada rendezvous and its implementation.

The Ada tasking concept contains several constructs and requirements that can make an implementation inefficient. The most important are:

- Task activation, termination and dependency.
- Rendezvous including timed entry calls and select statements.
- Queue handling.

Present implementations of the rendezvous are much more time consuming than for example a procedure call. This is due to the fact that the rendezvous is a complicated language construct which has several possible execution sequences involving two processes. This together with some asynchronous aspects makes the rendezvous hard to implement efficiently.

The memory model used is likely to have a major effect on how tasking are implemented. It also heavily influences the memory management scheme, which in Ada can become quite complicated. The memory model implied by Ada, the cactus stack, is described and shown to be in conflict with efficiency aspects for traditional memory organizations.

How the cactus stack is implemented is of course heavily dependent on the memory model chosen. This is one of the keys to good performance in a multiprocessor computer where shared memory can be physically distributed. In paper VI two different ways of implementing the Ada memory model are given.

The first model assumes that all tasks in one Ada program shares the same address space. The memory management is then handled by the run time system of that program. This can be done in two ways. One is to assign a fixed part of the memory for each task to use as stack space. Another way is to allocate and deallocate memory as needed, which leads to a complicated memory management.

The second model assumes one address space for each task. This model can't support the sharing needed by Ada directly. In paper VI a solution to this problem is presented using special hardware support and special run time code to detect and perform accesses to shared variables outside a task address space.

In order to support Ada efficiently certain restrictions to the Ada language can be made. These restrictions can be made by adopting a local coding policy or enforced by changing the language definition. Several areas can be identified for this: dynamic sharing of data on the cactus stack, no synchronous task activation and termination and simplified queue handling.

In many cases it is possible for the compiler to detect that only a subset of Ada is used and generate code to support this subset only. For example if no tasks can depend on a block then there is no need to support synchronous termination of dependent tasks for this block, i.e. it can left without checking that all its dependent tasks are terminated.

Some of the suggested improvements have been implemented, e.g. simplified queue management and generating run time support for only the actually used language constructs. The resulting improvements were measured and gave a 5 - 25 % decrease of execution time for different types of rendezvous.

To support full Ada really efficiently, hardware support is needed. In paper VI two types of smart queue memories are proposed that implements entry queues, the delay queue and the ready queue directly. A smart queue memory consists of an associative memory, an array of dynamic length FIFO queues and a control section.

It directly supports operations like insert in queue, remove from queue and get first from queue. It is estimated that such hardware support would result in at least an improvement by a factor of 2 - 4.

Acknowledgements

My advisor Lars Philipson got me started in this area and has guided and supported me through all these years. He has always been willing to discuss my work providing new insights, indicating new problems to attack thus keeping me busy. Lars has also taught me what I know about research methodology.

I would like to thank all members, past and present, of the MUMS research group who made this work possible and provided a stimulating environment. Fredrik Bergqvist, Björn Breidegard, Bo Nilsson and Per Stenström (the hardy boys) and Peter Molin, Kjell Persson, Joachim Roos and Peter Thulin (the soft guys). Thanks also to all my friends at CMU, Anita Jones and Paul Hilfinger in particular, for their help and support during my year there.

Lars together with Rolf Johannesson has provided me with an excellent research environment free from bureaucrats and administrative trivia for which I am immensely grateful. Bertil and Kerstin Andersen has kept me and my wife fit for fight thus giving us the necessary strength for the final finish.

This research was sponsored by the Swedish National board for Technical Development (STU) under contract numbers 80-3962, 83-3647 and 85-3899.

References

- A. Ardö and L. Philipson: "Evaluation of commercially available retargetable and rehostable Ada systems", Proceedings of the third Ada-Europe/AdaTEC Conference, June 1984.
- A. Jones and P. Schwarz: "Experience Using Multiprocessor Systems – A Status Report", Computing Surveys 12(2), June, 1980.
- K. V. Nori, U. Amman, K. Jensen, H. H. Nageli and C. Jacobi: "Pascal – the Language and its Implementation", John Wiley & Sons, 1981, Ch 9: Pascal-P Implementation Notes.
- L. Philipson, B. Nilsson and B. Breidegard: "A Communication Structure for a Multiprocessor Computer with Distributed Global Memory", Proceedings of the 10th Annual Symp. on Computer Architecture, pp 334-340, 1983.
- L. Philipson: "VLSI based design principles for MIMD multiprocessor computers with distributed memory management", Proceedings of the 11th Annual Symposium on Computer Architecture, June 1984, pp 319-327.
- P. Stenström and L. Philipson: "A layered test-bed for experimental evaluation of design principles for MIMD multiprocessors with shared memory", Technical report, Department of Computer Engineering, University of Lund, November 1985.

We shall have to evolve
problem-solvers galore -
since each problem they solve
creates ten problems more.

Piet Hein

